Week 1 Notes

Lectures 1 & 2

University of Massachusetts Amherst, CS389

1 Supervised Learning (lecture 1)

Definition 1.1 (Machine Learning)**.** A program, p, is able to perform a task, t, better according to some metric, m, when given experience, e. (Mitchell, 1997)

Definition 1.2 (Model)**.** A *model* is some structure that uses parameters to define a function. An example of a model is $y = wx + b$, which is a linear model. In another more explicit notation, we might write a linear model as (refer to subsection 1.1 for more info):

$$\hat{y} = \sum_{j=1}^{m} w_j \cdot X_j + b = XW^T + b, \tag{1.1}$$

where we have a weight w_j for each feature X_j, a bias term b, and an output vector $\hat{y}$.

The intuition is that, the parameters (i.e. W and b) can be changed based off the data so that the model performs the optimal function/prediction/output. We say that the model learns because it is able learn the right combination of W and b to give an optimal output (global optimality is not guaranteed; we will talk more about this in the next lecture). The weights, w_j, determines how important a feature x_j is to the $\hat{y}$. When the parameters of a machine learning model are fixed, the model can be viewed as a function, F, of input features X:

$$\hat{y} = F(X_1, ..., X_m) = w_1X_1 + ... + w_mX_m + b. \tag{1.2}$$

For example, the inputs of a weather model can be the humidity, temperature, air speed, etc., and the output could be *how much* it will rain (a type of model known as a *regression model*) or a probability distribution of *what* the weather will be like (a type of model known as a *classification model*). In a regression model the output $y \in \mathbb{R}$ while the output of a classification model is a probability distribution $Y \sim p(x)$. A classification model with 2 classes is a binary classification model, and a model with 2 or more classes is a multi-class classification model.

Definition 1.3 (Supervised Learning)**.** A *supervised learning* system is a system in which we can train a model with labeled data. For example, given an image x_1 the label y_1 could be "dog". A collection of these is a *dataset*, D:

$$D = \{(x_1, y_1), ..., (x_n, y_n)\}, \tag{1.3}$$

where x_i is the input vector to the i^{th} sample and y_i is the corresponding label to the i^{th} sample. The goal in supervised learning is to learn a function F such that for a new pair of data (x, y), we have $F(x) \approx y$ with high probability [3].

1.1 Why $\hat{y} = XW^T + b$?

Naively, we can implement a linear model using Algorithm 1. However, this is a very ugly implementation and for-loops in Python are slow. For our implementation, it would be better to express the operation using matrices. Generally we want X to have shape $(n \times m)$ where n is the number of data and m is the number of features (i.e each row/data has m features), therefore a $(m \times n)$ shape would not make sense. We also want the output to have shape $(n \times p)$ where p are the predictions

Algorithm 1 Naive Linear Model

1: $y = b$
2: **for** j in range(m) **do**
3: $\quad y \mathrel{+}= w_j * x_j$ ▷ scalar multiplication
4: **end for**

(i.e. each row/data has p predictions). The problem with $W^T X$ is that we need to transpose the X matrix which does not make sense to do so:

$$(?, ?) \times (n, m) \to (n, p). \tag{1.4}$$

Instead, writing it as XW^T makes more sense:

$$(n, m) \times (?, ?) \to (n, p), \tag{1.5}$$

where the weight matrix can now have shape (m, p). However, we want to represent the weight matrix as shape (p, m) as it is equivalent to saying "a prediction p can be made if given m features". Therefore, the dimensions of X, W, and output will be:

- $W \to (p, m)$
- $X \to (n, m)$
- Out $\to (n, p)$

We can see this convention being adopted by Pytroch (torch.nn.Linear)[1]. Another reason to represent W and X as matrices is that this form is easier to differentiate for gradient descent.

2 Regression (lecture 1)

Definition 2.1 (Linear Regression)**.** If we have a linear model (i.e. $y = XW^T + b$) finding the particular parameters (W and b) for this linear function is called *linear regression*. You may recall procedures for finding the line of best fit from a previous course - this is the same process, in an arbitrary dimension.

3 Loss Function (lecture 2)

Definition 3.1 (Loss Function)**.** A *loss function* measures how *poorly* the model is performing. The lower the loss, the better. A loss of zero means it makes perfect predictions. An example of a well-known loss function is the *Mean Absolute Error* (MAE) and can be written formally as:

$$\text{MAE}(y, \hat{y}') = \frac{1}{N} \sum_{n}^{N} y_n - \hat{y}'_n, \tag{3.1}$$

where y_n is the n^{th} label in the data and $\hat{y}'_n$ is the output of the model when given the n^{th} data, x_n. Another loss function is the *Mean Square Error* (MSE) and can be written formally as:

$$\text{MSE}(y, \hat{y}') = \frac{1}{N} \sum_{n}^{N} (y_n - \hat{y}'_n)^2. \tag{3.2}$$

The MAE will prefer outliers while MSE will prefer none to be extremely far. It is common practice to normalize the loss by the total number of training samples, N, so that the output can be interpreted as the average loss per sample [3].

[1] https://pytorch.org/docs/master/generated/torch.nn.Linear.html

3.1 Properties of the Loss Function

1. The minimal loss value should correspond to the line of best fit
2. The loss must be defined for all outputs and labels (no divide by 0)
3. The loss function should be differentiable (for gradient descent)

Therefore, an ideal loss function, $L(w)$, is convex, continuous, and differentiable.

4 Gradient Descent (lecture 2)

In the previous section, we described two common loss functions that describes how poor the model is performing. Our goal is to find the optimal set of parameters that minimizes the loss:

$$W^* = \arg\min_W L(W) \tag{4.1}$$

Clearly, this is an optimization task. The most common tool for finding the optimum in a problem such as this is known as *gradient descent*. There are other ways of performing optimization (i.e. LBFGS) but gradient descent is the most common and established way of optimizing Neural Network loss functions [4].

Definition 4.1 (Gradient of Loss)**.** The *gradient of loss*, ∇Loss, is the *direction of steepest ascent*. The negative gradient of loss, $-\nabla$Loss, is the direction of steepest descent. The gradient of loss with respect to the weight can be written as:

$$\nabla_W \text{Loss} = \frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, ..., \frac{\partial L}{\partial W_m} \right) \tag{4.2}$$

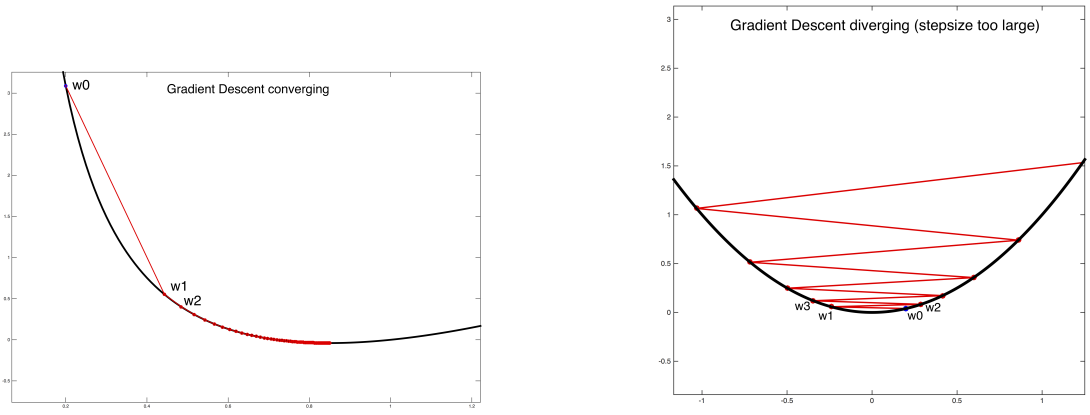
Definition 4.2 (Gradient Descent)**.** Is an optimization algorithm that minimizes the loss $L(W)$. The intuition behind the *gradient descent* algorithm is to iteratively update W_i in the direction of the negative gradient (steepest descent) until we converge to the value W_i that is minimized by $L(W_i)$, which can be denoted as W_i^*. Taking a small enough step, α, in the negative gradient is guranteed to reduce loss and convergence to an optimal model. The step size, α, is also known as the *learning rate* is a hyperparameter. Formally, we can write all of this as[2]:

$$W = W - \alpha \nabla_W \text{Loss} \tag{4.3}$$

$$W_i = W_i - \alpha \frac{\partial L}{\partial W_i} \tag{4.4}$$

Whether we converge to an optimal solution depends on how we set the learning rate α. The model is unable to converge if α is too large, and is unable/expensive to converge because α is too small. This is visualized in Figure 1. Usually $\alpha > 0$ and is some small number. We say the model converges when we see that $\frac{\partial L}{\partial w}$ is minimized ($= 0$) or does not change after each iteration of the gradient descent algorithm.

[2] Assume that this model has no bias (i.e. $b = 0$) for simplicity



(a) α is just right

(b) α is too large

Figure 1: Consequence of choosing α [3]

4.1 Batch, Mini Batch, and Stochastic Gradient Descent

Definition 4.3 (Epoch)**.** An epoch is one complete pass of the training dataset

Definition 4.4 (Batch Gradient Descent)**.** Also known simply as gradient descent, describes a method that updates the parameters (W and b) using the *whole training dataset per epoch*. For example, one would use the mean of the gradients of all the training data to update the parameters. However it is wasteful to compute the loss function over the entire training set in order to perform a parameter update only once [4].

Definition 4.5 (Stochastic Gradient Descent)**.** Also known as on-line gradient descent, describes a method that updates the parameters after every *single* training data. When people refer to SGD, they usually refer to minibatch gradient descent approach as it is more commonly used. The stochastic gradient descent is technically a minbatch gradient descent with a batch size of 1.

Definition 4.6 (Minibatch Gradient Descent)**.** Is a method that updates the parameters on every subset with a batch size, B, of the training dataset per epoch. The batch size is a hyperparameter which are chosen because of memory constraints. The batch size is usually in powers of 2 because vectorized operations work faster when the inputs is the power of 2 [4].

References

[1] Cooper.

[2] Mengye Ren and Matthew MacKay, UoT CSC 411, Winter 2019. https://web.archive.org/web/20190513201539/https://www.cs.toronto.edu/~mren/teach/csc411_19s/lec/lec06.pdf

[3] Kilian Q. Weinberger, Cornell CS4780, Fall 2018. https://web.archive.org/web/20180827054501/https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote01_MLsetup.html https://web.archive.org/web/20181016051558/https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote07.html

[4] Fei-Fei Li, Jiajun Wu, and Ruohan Gao, Stanford CS231n, Spring 2022. https://web.archive.org/web/20230109135558/https://cs231n.github.io/optimization-1/